(4)

BBN Systems and Technologies Corporation

A Subsidiary of Bolt Beranek and Newman Inc.

AD-A214 585

Report No. 7142

ACCESS TO MULTIPLE UNDERLYING SYSTEMS IN JANUS

Philip Resnik

September 1989

Submitted by:

BBN Systems and Technologies Corporation
10 Moulton Street
Cambridge, MA 02138

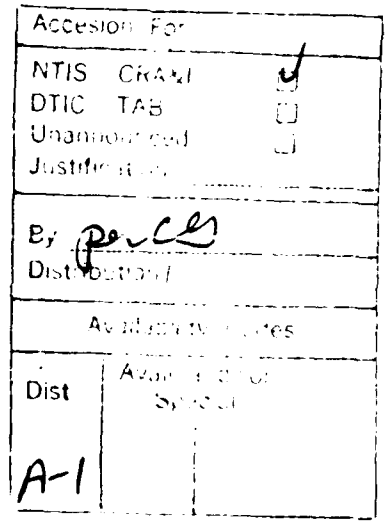
Submitted to:

Defense Advanced Research Projects Agency (DARPA)
1400 Wilson Blvd.
Arlington, VA 22209

Report No. 7142

ACCESS TO MULTIPLE UNDERLYING SYSTEMS IN JANUS

Philip Resnik



September 1989

Submitted by:

BBN Systems and Technologies Corporation
10 Moulton Street
Cambridge, MA 02138

Submitted to:

Defense Advanced Research Projects Agency (DARPA)
1400 Wilson Blvd.
Arlington, VA 22209

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by ONR under Contract No. 00014-85-C-0016. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 7142	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION BBN Systems and Technologies Corporation 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 10 Moulton Street Cambridge, MA 02138	7b. ADDRESS (City, State, and ZIP Code) Department of the Navy Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Defense Advanced Research Projects Agency 8b. OFFICE SYMBOL (If applicable) DARPA ISTO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Access to Multiple Underlying Systems in Janus

12. PERSONAL AUTHOR(S)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim Technical Rept	FROM _____ TO _____	1989, September	34

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Janus, Irus, Code Generation, Natural Language Interfaces

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The job of the back-end of any natural language interface is to translate a logical description of what the user wants (a *request*) into an efficient plan for fulfilling that request. Typically the request is to produce data fro some *underlying system*; that is, the database, applications program, or other system with which the user is communicating by means of the interface. There has been a fair amount of work on the problem of natural language interfaces to single underlying systems.

As computer systems become more complex, there is more opp.. ity for combining the strengths of more than one system in order to perform a task. For example, on might imagine combining several resources: a database for storing relational information with an applications program to perform calculations based on that information, an expert system to perform inferences, and a display system to present data in a useful way. In such an environment a "seamless" natural language interface can become a very effective tool, allowing the user to retrieve and manipulate information without needing to pay attention to the details of any particular resource.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

The back-end of such an interface, however, is necessarily more complex: not only must it be able to translate the user's request into executable code, but it must also be capable of organizing the various resources at its disposal, choosing which combination of resources to use, and supervising the transfer of data among them. We call this the *multiple underlying systems* (MUS) problem. This document describes one approach to the MUS problem, a *MUS component* implemented as part of the back end of the Janus natural language interface.

Table of Contents

1 Introduction

The job of the back-end of any natural language interface is to translate a logical description of what the user wants (a *request*) into an efficient plan for fulfilling that request. Typically the request is to produce data from some *underlying system*; that is, the database, applications program, or other system with which the user is communicating by means of the interface. There has been a fair amount of work on the problem of natural language interfaces to single underlying systems, for example, [5].

As computer systems become more complex, there is more opportunity for combining the strengths of more than one system in order to perform a task. For example, one might imagine combining several resources: a database for storing relational information with an applications program to perform calculations based on that information, an expert system to perform inferences, and a display system to present data in a useful way. In such an environment a "seamless" natural language interface can become a very effective tool, allowing the user to retrieve and manipulate information without needing to pay attention to the details of any particular resource.

The back-end of such an interface, however, is necessarily more complex: not only must it be able to translate the user's request into executable code, but it must also be capable of organizing the various resources at its disposal, choosing which combination of resources to use, and supervising the transfer of data among them. We call this the *multiple underlying systems* (MUS) problem. This document describes one approach to the MUS problem, a *MUS component* implemented as part of the back end of the Janus natural language interface.

We begin in section 2 with a brief description of Janus' *type system*, a component of the semantic interpretation language (WML, for World Model Language) that plays an important role in both front-end and back-end processing. Section 3 describes the translation of WML expressions into a simplified, normalized form. Section 4 discusses the way that the system represents the capabilities of underlying systems. Section 5 describes the algorithm used for finding an effective combination of the services provided by the underlying systems, in order to satisfy a given request. Section 6 describes how this combination of resources is used to produce an *execution plan*, and section 7 deals with the execution of this plan and the transfers of data this often entails. Finally, section 8 discusses the implementation of this approach, as well as limitations and extensions to this work.

2 The Type System

The syntax of WML is "modelled after languages of the typed lambda calculus" ([2], p. 27). The import of this is perhaps most concisely expressed in [6]:

> Each type expression [or *type*] is associated with a set of entities or structures which is termed its *domain*. Every expression of the language... can be mapped to a type expression, whose domain serves to delimit the range of values the expression can take on.

A complete description of the type system associated with WML is beyond the scope of this document, but in this section we attempt to convey a sense of it by means of examples.

The set of types has two major subsets -- those that are independent of the domain, and those that are specific to the domain. The former set includes the types TV (truth-value), INTEGERS, STRINGS, REALS, TIMES, and WORLDS; the latter consists of types constructed from concepts and roles in the domain-model. For example, if *VESSEL* and *LOCATION* are domain-model concepts, and *SHIP-LOCATION* and *COMBAT-READY* are domain roles, then some related types might include:

Type	Denotation
VESSEL	all vessels
(S VESSEL)	all *sets* of vessels
(TUPLE VESSEL LOCATION)	all ordered pairs of vessel, location
(S (TUPLE VESSEL LOCATION))	all sets of such ordered pairs
(FUNC-TYPE VESSEL LOCATION)	functions from vessels to locations
(FUNC-TYPE VESSEL TV)	unary predicates on vessels

Here we present some types together with examples of logical subexpressions having those types (that is, *(TYPEOF expression) = type*).

1. type VESSEL
```
Vincennes
(IOTA ?JX1 VESSEL (COMBAT-READY ?JX1))
```
2. type (S VESSEL)
```
(SETOF Vincennes Kennedy)
(POWER VESSEL)
(SET ?JX2 VESSEL (COMBAT-READY ?JX2))
(IOTA ?JX1 (POWER VESSEL) (COMBAT-READY ?JX1))
```
3. type (FUNC-TYPE VESSEL LOCATION)
```
SHIP-LOCATION
(LAMBDA (?JX3) VESSEL (VESSEL-LOCATION ?JX3))
```
4. type (FUNC-TYPE VESSEL TV)
```
COMBAT-READY
(LAMBDA (?JX4) VESSEL
  (AND (EQUAL (VESSEL-LOCATION ?JX4) "HAWAII")
       (COMBAT-READY ?JX4)))
```

3 Normalizing WML Expressions

WML input expressions are simplified and normalized before they are further processed by the multiple underlying systems (MUS) component. This simplification process has five stages:

1. the extensionalization of intensional subexpressions,
2. the translation of the entire expression into a modified disjunctive normal form,
3. the elimination of unnecessary equivalences,

4. the application of underlying-system-independent rewrites, and

5. the use of printfunctions for improving responses.

These stages occur sequentially, and -- as the system is currently implemented -- must be done in the order presented here. In this section, we discuss each of these stages in some detail.

3.1 Extensionalizing Intensional Subexpressions

The first stage in normalizing a WML expression is the extensionalization of the logical form's intensional subexpressions. Most underlying systems, whether they are databases, expert systems, or other applications programs, are extensional. Those that do take time and/or possible worlds into account tend to do it in a very discrete fashion. For example, the Navy's Integrated Data Base (IDB) divides the temporal continuum for ships' combat-readiness ratings into previous-readiness, current-readiness, and projected-readiness; a BBN object-oriented simulator supports hypothetical worlds, but maintains discrete world-states in a tree-like hierarchy.

This discreteness makes it possible to "extensionalize" many intensional subexpressions (ideally all of them!) in the following manner: given an intensional context (i.e., an intension together with its time and world indices), every predicate within the intensional expression "absorbs" the time/world information, either (1) by replacing it with a related but time- (or world-) specific predicate, or (2) by adding temporal/world-related information to the predicate's argument list.

For example, the expression

```
((INTENSION (SHIP-LOCATION ?JX1 ?JX2)) 13-JULY-1965 DEFAULT-WORLD)
```

might, according to the first of these methods, be extensionalized as

```
(PAST-SHIP-LOCATION ?JX1 ?JX2)
```

assuming that *PAST-SHIP-LOCATION* is a domain model predicate and that there is appropriate machinery for choosing it on the basis of the time index. This approach loses information, of course -- such a process of extensionalization could not take place unless it is certain that no underlying system would be interested in the specific time index, that is, that the new predicate (*PAST-SHIP-LOCATION*) is sufficient for the purposes of any underlying system.

Alternatively, the indexical information could be absorbed into the predicate argument structure, as in

```
(SHIP-LOCATION-AT-TIME ?JX1 ?JX2 13-JULY-1965) .
```

This approach preserves the information: underlying systems are free to use the time argument, to infer from it a predicate like *PAST-SHIP-LOCATION*, or to ignore it altogether.

At the time of this writing, the implementation of extensionalization is incomplete, and for the moment assumes that no predicate cares about time or world indices. Thus the intensional expression would simply be extensionalized as

```
(SHIP-LOCATION ?JX1 ?JX2) .
```

The process of extensionalization, as characterized here, applies to more complex intensions, as well: one may do a recursive walk through an intension, rewriting predicates (as above, for example) on the basis of the time and world indices.

3.2 Disjunctive Normal Form

The second stage of normalization is the translation of the extensionalized WML expression into a somewhat simplified logical expression in a modified disjunctive normal form (DNF).

The expression is translated into a disjunctive normal form for two main reasons. We normalize the expression (reducing the number of embedded subexpressions, for example) in order to simplify the process of matching various pieces of it to underlying system capabilities. We choose to use a disjunctive normal form because:

- *In the simplest case, an expression in disjunctive normal form is simply a conjunction of clauses, a particularly easy logical form to cope with.*
- Even when there are disjuncts, each can be individually handled as a conjunction of clauses, and the results then combined together via union, and
- Bringing disjunctions to the top level allows patterns to match in many cases where it would otherwise not be possible. For example, given the (non-normalized) pattern

```
(AND  (OR  (IN.CLASS ?JX1 SUBMARINE)
           (IN.CLASS ?JX1 AIRCRAFT))
      (LENGTH ?JX1 ?JX2))
```

a service seeking to match the pattern

```
(AND (IN.CLASS <x> SUBMARINE)
     (LENGTH <x> <y>))
```

could not match. The DNF, on the other hand,

```
(OR  (AND (IN.CLASS ?JX1 SUBMARINE)
          (LENGTH ?JX1 ?JX2))
     (AND (IN.CLASS ?JX1 AIRCRAFT)
          (LENGTH ?JX1 ?JX2)))
```

allows the match to take place, by keeping the relevant information together. In a disjunctive normal form, each disjunct effectively carries all the information necessary for a distinct subquery.

A standard disjunctive normal form is a disjunction of conjunctions of predicates or negated predicates; no variables in such an expression are explicitly quantified, and all are assumed to be implicitly universally quantified. Existentially quantified variables have been replaced by skolem terms denoting some individual instantiation of the variable.

The modified disjunctive normal form differs from a standard DNF in several respects:

- There is a *response clause* for every query; that is, an additional predicate whose arguments are the variables for which we want returned values. In a query requesting a set of objects (e.g. "Which ships are in the Indian Ocean?") the argument in the response clause will be the variable denoting the set in question. The same is true for queries requesting individuals (e.g. "the ship whose speed is 30 knots"); the resulting logical form will seek *all possible* individuals that meet the same description.

In a yes/no or existential query, the response clause will contain all variables in the query, since any instantiation of all the query variables means an affirmative answer; the inability to find any such instantiation means an answer in the negative. In such cases, a particular variation on the response predicate is used: rather than using the special predicate *RESPONSE*, we use the special predicate *VALUE-EXISTS-RESPONSE* instead; this preserves the information that the query's intent is to find out if values exist, rather than to have them returned.[1]

- All functional terms must appear as predicates: if P is a binary predicate and Q is a unary function, then *P(x, Q(y))* must appear as *P(x, z) and Q'(y, z)*, where Q'(y, z) is true iff Q(y)=z. For example, the clause

```
(GREATER-THAN (SPEED-OF ?JX1) 30)
```

  will appear as

```
([AND] (SPEED-OF' ?JX1 ?JX2)
       (GREATER-THAN ?JX2 30))
```

- Similarly, provision is made for complex terms like database aggregates: for example, cardinality, average, and sum. Such complex terms may only appear as the first argument to a special predicate called *IS-TERM*; the second argument is always a variable that represents the term. For example, if the logical expression asks for the cardinality of the ships in the Indian Ocean, we would use the following clause:

```
(IS-TERM #$(CONTEXT
             :OPERATOR CARDINALITY
             :OPERATOR-VAR ?JX2
             :CLASS-EXP
               ((IN.CLASS ?JX2 SHIP)
                (SHIP-LOCATION ?JX2 "INDIAN OCEAN")))
           ?JX1)
(RESPONSE ?JX1)
```

- There is no implicit assumption of universal quantification for unquantified variables -- expressions in the modified DNF may contain universal quantification.

- Existential quantifiers are removed, not by replacing existentially quantified variables with skolem terms, but simply by removing the explicit existential quantification. The resulting unquantified variables, along with all other unquantified variables in the form, are considered to be *query quantified*.

  The term *query quantified* refers to variables for which we would like to get all possible instantiations. Such variables are neither existentially quantified (since we're interested in all instantiations) nor universally quantified (since universal quantification has no notion of returning values); this kind of quantification is more like that of the variables in a PROLOG expression.

  Notice that, because universal quantifications are not removed, there is no need to skolemize existentially quantified variables appearing within the scope of universal quantifiers.

The following is a specification of the modified disjunctive normal form. Square brackets ([]) indicate optional elements. Contexts are objects with internal components (implemented as LISP structures) -- these objects represent distinct logical environments whose internal components must be kept separate from the remainder of the expression.

expression :- ([AND] clause^{+} [response-clause])

[1] A cooperative system may still return the values if they exist; for example, "Are any ships C1?" might lead to the response *Yes, the C1 ships are*...

```
clause          :-      (predicate arg+)     |
                        context-clause         |
                        is-term-clause         |
                        in-class-clause

response-clause :-      (RESPONSE var+) | (VALUE-EXISTS-RESPONSE var+)

context-clause[2]  :-  disjunction-context |
                        negation-context     |
                        quantifier-context

disjunction-context :-  #S(CONTEXT
                          :OPERATOR      OR
                          :CLASS-EXP     (clause+)
                          [:FREE-VARS    (var+)]
                          [:LOCAL-VARS   (var+)]      )


negation-context :-     #S(CONTEXT
                          :OPERATOR      NOT
                          :CLASS-EXP     clause
                          [:FREE-VARS    (var+)]
                          [:LOCAL-VARS   (var+)]      )

quantifier-context :-   #S(CONTEXT
                          :OPERATOR      FORALL
                          :OPERATOR-VAR  var
                          :CLASS-EXP     expression
                          :CONSTRAINT    expression
                          [:FREE-VARS    (var+)]
                          [:LOCAL-VARS   (var+)]      )

is-term-clause  :-      (IS-TERM term-context var)

term-context :-         #S(CONTEXT
                          :OPERATOR      term-context-operator
                          :OPERATOR-VAR  var
                          :CLASS-EXP     expression
                          :CONSTRAINT    expression
                          :STAT-VAR[3]   var
                          [:FREE-VARS    (var+)]
                          [:LOCAL-VARS   (var+)]      )

in-class-clause :-      (IN.CLASS var simple-type)

arg             :-      var | constant

var             :-      ?JX1 | ?JX2 | ?JX3 | ...
```

[2]Notice that the contents of a context object depends upon the operator: for example, in a quantifier context (e.g. FORALL), the class-expression field (class-exp) is an expression, whereas within a negation context, that field contains a clause.

[3]This field is not currently used, but is intended to provide a place to store an additional variable, if the syntax of the operator requires it.

```
term-context-operator   :- CARDINALITY | AVG | SUM | ...
simple-type      :-     domain-model-concept-name | type-system-atomic-type
predicate        :-     domain-model-role-name | type-system-func-type
constant         :-     type-system-individual | string | number
```

3.3 Eliminating Equivalences

Occasionally, a logical expression will include unnecessary equivalences between terms -- equating a constant with a variable, for example, and then using the variable elsewhere in the expression where the constant would do just as well. It is helpful to eliminate such equivalences early in processing.

Consider the following expression, resulting from the query "What are the readinesses of the cruisers that are not C1?":

```
(AND (EQUAL ?JX109 ?JX110)
     (IN.CLASS ?JX111 CRUISER)
     (VESSEL-OVERALL-READINESS-OF ?JX111 ?JX110)
     #S(CONTEXT :OPERATOR NOT
                :FREE-VARS (?JX110)
                :LOCAL-VARS NIL
                :OPERATOR-VAR NIL
                :CLASS-EXP (EQUAL ?JX110 C1)
                :STAT-VAR NIL
                :CONSTRAINTS NIL)
     (IN.CLASS ?JX110 READINESS-RATING)
     (RESPONSE ?JX109))
```

Here the variables ?JX109 and ?JX110 are equated; the elimination of that equivalence simplifies the expression without changing its meaning. The resulting expression is

```
(AND (IN.CLASS ?JX111 CRUISER)
     (VESSEL-OVERALL-READINESS-OF ?JX111 ?JX207)
     #S(CONTEXT :OPERATOR NOT
                :FREE-VARS (?JX207)
                :LOCAL-VARS NIL
                :OPERATOR-VAR NIL
                :CLASS-EXP (EQUAL ?JX207 C1)
                :STAT-VAR NIL
                :CONSTRAINTS NIL)
     (IN.CLASS ?JX207 READINESS-RATING)
     (RESPONSE ?JX207))
```

Notice that a new variable, ?JX207, has replaced the equivalence class {?JX109, ?JX110}.

Not all statements of equality are unnecessary; for example, the query "Are there (exactly) three C1 cruisers?" results in the following expression:

```
(AND (VALUE-EXISTS-RESPONSE ?JX210)
     (IN.CLASS ?JX210 CRUISER)
     (IS-TERM
      #S(CONTEXT :OPERATOR CARDINALITY
                 :FREE-VARS NIL
                 :LOCAL-VARS NIL
                 :OPERATOR-VAR ?JX210
                 :CLASS-EXP NIL
                 :STAT-VAR NIL
                 :CONSTRAINTS NIL)
      ?JX219)
     (EQUAL ?JX219 3)
     (VESSEL-OVERALL-READINESS-OF ?JX210 C1))
```

Here the clause *(EQUAL ?JX219 3)* is not an equivalence that can be eliminated.

3.4 System-independent Rewrites

At this stage of the normalization process, the system permits the application of obligatory rewrite rules. These rules must be independent of the underlying systems: *both pattern and result must consist of domain-model* information, and they may not contain any references to structures or data in the underlying system(s).

Rewrite patterns may seek to match both simple clauses (i.e., those that are not contexts), and context-clauses. Similarly, results may be either contexts or simple clauses. For example, the following rewrite might be used if it was known that the number of subordinates of a manager corresponded to the number of employees in a manager's department:

```
(define-simple-rewrite[4]
  :pattern ((in.class x manager)
            (:context
             :operator CARDINALITY
             :class-exp ((in.class y person)
                         (subordinate-to y x))))
  :result  ((in.class x manager)
            (department-of x z)
            (employee-count z y)))
```

By using this rewrite rule, we transform a query in which one actually counts elements in a set (via the cardinality term) into one in which a single table lookup is used instead.

[4] In simple rewrites, x, y, z, and w are always variables.

3.5 Printfunctions

Often the logical content of a query does not reflect its desired interpretation. For example, a query as simple as "List the cruisers," if interpreted literally, produces a listing of the database's *internal representation* for each cruiser. In the Navy's IDB domain, this representation is a number called an IUID -- a number that is almost certain to be completely useless to the user as a means of ship identification. What one would *really* like is for the system to be smart enough to interpret the question as "List the *names* of the cruisers." *Printfunctions* provide just that functionality.

The printfunction machinery is quite simple. With certain classes of objects (e.g., the domain-model concept *VESSEL*) one associates a specification for how members of that class should be presented to the user, called a *printfunctions list*. Each element of the printfunctions list (each *printfunction*) is either (1) the name of a domain-model role, or (2) the special symbol :IDENTITY. As a postprocessing step of the normalization, the variables on the response list are examined, and a *new* response list created as follows:

```
Let the new response list begin as an empty list
For each variable v on the original response list
  Let type be the variable's type
  If type has no printfunction list associated with it
     Add v to the new response list
  Else
     For each element pfn in the type's printfunction list
       If pfn is :IDENTITY
          Add v to the new response list
       Else pfn is a domain model role:
          Let new be a new variable
          Add the clause (pfn v new) to the query itself
          Add the variable new to the new response list
```

Printfunctions are inherited -- in the examples in the following section, responses involving ship classes like cruiser and aircraft-carrier are always expressed as responses involving the names of the ships because the class *VESSEL* (the top-level class for ships) has the printfunction list *(NAMEOF)* associated with it.

3.6 Examples of Normal Form

The examples in this section consist of a WML expression, followed by its DNF, and then followed by the normalized form after rewrites and printfunctions have applied (if there was any change).

1. "List the ships."

```
(BRING-ABOUT
  ((INTENSION
     (EXISTS ?JX1 LIST
        (OBJECT.OF ?JX1 (IOTA ?JX2 (POWER VESSEL) T))))
   TIME WORLD))

Normalized expression:

(AND (IN.CLASS ?JX1 LIST)
     (IN.CLASS ?JX2 VESSEL)
     (OBJECT.OF ?JX1 ?JX2)
     (MEMBER ?JX1 ?JX3)
     (IN.CLASS ?JX3 (POWER EVENT))
     (RESPONSE ?JX3))

Expression after rewrites and printfunctions have applied:

((RESPONSE ?JX70)
 (NAMEOF ?JX2 ?JX70)
 (IN.CLASS ?JX2 VESSEL))
```

2. "Which ships are C1?"

```
(QUERY
  ((INTENSION
     (PRESENT
       (INTENSION
        (IOTA ?JX4 (POWER VESSEL)
          (VESSEL-OVERALL-READINESS-OF ?JX4 C1)))))
   TIME WORLD))

(AND (IN.CLASS ?JX4 VESSEL)
     (VESSEL-OVERALL-READINESS-OF ?JX4 C1)
     (RESPONSE ?JX4))

((RESPONSE ?JX72)
 (NAMEOF ?JX4 ?JX72)
 (IN.CLASS ?JX4 VESSEL)
 (VESSEL-OVERALL-READINESS-OF ?JX4 C1))
```

3. "Which cruisers are not C1?"

```
(QUERY
  ((INTENSION
     (PRESENT
       (INTENSION
         (IOTA ?JX8 (POWER CRUISER)
         (NOT (VESSEL-OVERALL-READINESS-OF ?JX8 C1))))))
   TIME WORLD))

(AND
  (IN.CLASS ?JX8 CRUISER)
  #S(CONTEXT
      :OPERATOR NOT
      :FREE-VARS (?JX8)
      :LOCAL-VARS NIL
      :OPERATOR-VAR NIL
      :CLASS-EXP (VESSEL-OVERALL-READINESS-OF ?JX8 C1)
      :STAT-VAR NIL
      :CONSTRAINTS NIL)
  (RESPONSE ?JX8))

((RESPONSE ?JX73)
 (NAMEOF ?JX8 ?JX73)
 (IN.CLASS ?JX8 CRUISER)
 #S(CONTEXT
    :OPERATOR NOT
    :FREE-VARS (?JX8)
    :LOCAL-VARS NIL
    :OPERATOR-VAR NIL
    :CLASS-EXP (VESSEL-OVERALL-READINESS-OF ?JX8 C1)
    :STAT-VAR NIL
    :CONSTRAINTS NIL))
```

4. "Are any carriers harpoon capable?"

```
(QUERY
  ((INTENSION
     (PRESENT
       (INTENSION
         (EXISTS ?JX20 (POWER AIRCRAFT-CARRIER)
           (HARPOON-CAPABLE-VESSEL ?JX20)))))
   TIME WORLD))

(AND (VALUE-EXISTS-RESPONSE ?JX20)
     (IN.CLASS ?JX20 AIRCRAFT-CARRIER)
     (HARPOON-CAPABLE-VESSEL ?JX20))

((VALUE-EXISTS-RESPONSE ?JX74)
 (NAMEOF ?JX20 ?JX74)
 (IN.CLASS ?JX20 AIRCRAFT-CARRIER)
 (HARPOON-CAPABLE-VESSEL ?JX20))
```

5. "Are the cruisers and the carriers c1?"

```
(QUERY
  ((INTENSION
     (PRESENT
       (INTENSION
         (VESSEL-OVERALL-READINESS-OF
           (SETOF
             (IOTA ?JX56
               (POWER
                 (SET-TO-PRED
                   (IOTA ?JX59 (POWER CRUISER) T)))
               T)
             (IOTA ?JX57 (POWER AIRCRAFT-CARRIER) T))
           C1))))
   TIME WORLD))

(#S(CONTEXT
     :OPERATOR OR
     :FREE-VARS (?JX64)
     :LOCAL-VARS (?JX56 ?JX59 ?JX57)
     :OPERATOR-VAR NIL
     :CLASS-EXP
     ((AND (EQ ?JX64 ?JX56)
           (IN.CLASS ?JX56 CRUISER))
      (AND (EQ ?JX64 ?JX57)
           (IN.CLASS ?JX57 AIRCRAFT-CARRIER)))
     :STAT-VAR NIL
     :CONSTRAINTS NIL)
 (VESSEL-OVERALL-READINESS-OF ?JX64 C1)
 (VALUE-EXISTS-RESPONSE ?JX64))
```

6. "How many cruisers are in the Indian Ocean?"

```
(QUERY
  ((INTENSION
     (PRESENT
       (INTENSION
         (CARD (IOTA ?JX65 (POWER CRUISER)
                 (IN.PLACE ?JX65 INDIAN.OCEAN))))))
   TIME WORLD))

(AND (IS-TERM
       #S(CONTEXT
           :OPERATOR CARDINALITY
           :FREE-VARS NIL
           :LOCAL-VARS NIL
           :OPERATOR-VAR ?JX65
           :CLASS-EXP ((IN.CLASS ?JX65 CRUISER)
                       (IN.PLACE ?JX65 INDIAN.OCEAN))
           :STAT-VAR NIL
           :CONSTRAINTS NIL)
       ?JX69)
     (RESPONSE ?JX69))
```

4 Servers and Services

In an environment with multiple underlying systems, one must have a uniform way to describe the capabilities of each underlying system. We adopt terminology similar to that of [3] and [4].

A *server* is a functional module typically corresponding to an underlying system or a major part of an underlying system. In the application of the MUS system being described here, there are two servers -- one named :ERL, which supports access to a relational database, and one called :LISP, which supports calls to arbitrary LISP functions. Each server has associated with it:

1. A number of *services*: objects describing a particular piece of functionality provided by a server. Specifying a service in MUS provides the mapping from fragments of logical form to fragments of underlying system code
2. An *execution planner*: a function that takes a piece of the *solution* to a query (see section 5) and builds from it a *partial execution plan* (see section 6)
3. An *executor*: a function that takes a partial execution plan together with input data, executes the plan, and produces output data (see section 7).

A service is an object consisting of the following components:

- **Name**: a symbol used to uniquely identify the service
- **Owner**: the name of the server to which this service belongs.
- **Cost**: a scalar value indicating the cost of this service; if unspecified, unit cost (1) is assumed.
- **Inputs**: a list of pattern variables, each of which has associated with it a name, a type, and a constraint. The type indicates the extent to which the input is optional: a type of *:GEN* indicates that input to this variable is optional, since this service can generate values for the variable; a type of *:TEST* indicates that input must be provided for the variable, since this service is only capable of applying some test to the input values; a type of *:TEST-ALL* indicates not only that input to this variable is obligatory, but that by the time the data for this variable reaches this service it must be filtered as completely as possible -- this is often the type for inputs to services that do response presentation, for example.

  The constraint associated with the variable is used for pattern-matching. The possible constraints include:

  1. *(symbol*)*: a list of symbols. Items matching this variable must be EQ to a symbol on the list.
  2. *(string*)*: a list of strings. Items matching this variable must be STRING= to a string on the list.
  3. *type*: a simple type. An item will match this variable if the type (i.e., type-system type -- see section 2) of the item is a subtype of *type*. (The type of a Janus variable *v* is *type* if the clause *(IN-CLASS v type)* appears in the (normalized) query.) This constraint does not pay attention to whether or not a type denotes a set -- if *type* is *(S SHIP)* (a set of ships), an item with type *SHIP* will match, and vice-versa.
  4. *function*: a function than takes one argument. An item *item* will match this variable if *(funcall function item)* returns a non-NIL value.
  5. *(SUBTYPE-OF type)*: a subtype specification. Items matching this variable must themselves be *types in the type system*; furthermore, they must be subtypes of *type*. For example, a variable with constraint *(SUBTYPE-OF SHIP)* would match *CRUISER* (since *CRUISER* is a subtype of

SHIP). Notice how this differs from (3), above: there the constraint is that *(SUBTYPE (***TYPEOF** *item) type)* must be true, whereas here the constraint is that *(SUBTYPE item type)* must be true.

6. *NIL*, *ANYTYPE*, *T*: these will match anything

- **Outputs**: a list of pattern variables, identifying the outputs of the service. Outputs need not have been inputs, nor must inputs to the service also be outputs.
- **Pattern**: a pattern specification which will match some piece of the logical form. The pattern specification must be a list, each element of which is the pattern for either a simple clause or for a context-clause. Within patterns, one can not have a variable predicate; however, the arguments to predicates *must* be pattern variables (see *inputs*, above, for a description of how to constrain what these variables may match).

  A pattern specification for a context-clause (*context-spec*) takes one of two forms.

```
(:context :operator operator
          [:free-vars (var+)]
          [:operator-var var]
          [:stat-var var]
          [:class-exp expression]
          [:constraints expression])
```

  For this form of pattern specification, the context-spec's operator must match the context's operator, and recursive calls to the matcher must return successfully for the :class-exp and :constraints.

```
(:context :operator operator
          [:covers-owner server-name])
```

  This second form of pattern specification allows one to say, "This service will match *any* context whose operator is *operator*, as long as there are solutions of the :class-exp subexpression and of the :constraints subexpression such that both solutions belong entirely to server *server-name*." For example, a context-spec for operator *CARDINALITY* specifying that it covers owner *:ERL* says, in effect, "This service can take the cardinality of any set, as long as that set can be obtained entirely by calls within the :ERL server." This is a useful method of providing general services that handle aggregate operations within a single server.

- **Method**: a code fragment or other information that the server will use in generating a partial execution plan from a solution that utilizes this service. What goes in the method slot depends entirely on the particular server to which the service belongs.

For fast access, services are indexed by the predicates in their pattern. That is, for every clause *(P x y)* in the pattern of some service S, there is a pointer from the symbol *P* to the service S. An exception to this is the *IN.CLASS* predicate: if a service's pattern includes *(IN.CLASS x C)*, the pointer will be from the symbol *C* rather than the symbol *IN.CLASS*; that is, the indexing proceeds as if the clause were *C(x)*.

As an example, consider the service-object corresponding to the :ERL server's ability to access a table associating ships with overall combat-readiness values:

```
NAME:       VESSEL-OVERALL-READINESS-OF859
OWNER:      :ERL
INPUTS:     (<x> <y>)
OUTPUTS:    (<x> <y>)
PATTERN:    ((VESSEL-OVERALL-READINESS-OF <x> <y>))
COST:       NIL[5]
METHOD:     (((VESSEL-OVERALL-READINESS-OF X Y))
              (BINDTOERL ((X IUID) (Y RDY)) IID.RDY))
```

The name and owner fields are straightforward: the service has a unique name and belongs to the server named :ERL (in a current implementation, :ERL is the server that can access the Navy's relational database). The pattern is also particularly simple, a single clause. Note that the variables printed as <x> and <y> are objects:

```
NAME:       X
TYPE:       :GEN
CONSTRAINT: VESSEL

NAME:       Y
TYPE:       :GEN
CONSTRAINT: READINESS-RATING
```

Because both are type :GEN, this service does not require input values for these variables. The pattern will match clauses only when the type of the first argument (matching <x>) is VESSEL, and the type of the second argument (matching <y>) is READINESS-RATING.

The method field for services belonging to the :ERL server contains two pieces: first, the pattern that was matched; second, a code-like fragment that relates variables to fields and specifies a table from which to draw those fields.[6]

The scheme for indexing services establishes a pointer from the symbol VESSEL-OVERALL-READINESS-OF to this service.

5 Formulation

The job of the formulation algorithm is to locate all services that might be resources for satisfying a request, and find the best possible combination of services from that set, where "best" typically means lowest-cost. This is inherently a search problem.[7] Previous approaches to the formulation problem have included using NIKL

[5]The cost field is unspecified, therefore this service is assumed to have unit cost.

[6]This is a simplification: the table specification may be a fragment of ERL code, complete with JOINs, SELECTs, etc.

[7]The formulation problem, when its input is a conjunction of non-negated simple clauses, can be viewed as a kind of set-covering problem (SCP), which is NP-complete [1]. The SCP can be formulated as follows: given a set $S = \{s_1, s_2, ..., s_n\}$ and a collection $C = \{C_1, C_2, ..., C_m\}$ such that each C_i is a (proper) subset of S and each C_i has a positive cost c_i, find the subset C' of C such that (1) the union over C' equals S, and (2) the sum of the costs over C' is minimized. In the formulation problem, S is the set of clauses, and each element of C is a service.

classification [4] and a kind of A* search [3]. The approach here resembles a beam search, and uses a greedy heuristic.

The first two subsections describe two objects, *partial solutions* and *matches*, that are important at the implementation level; the third section describes the formulation algorithm.

5.1 Partial Solutions

The main structure used in the formulation stage is the *partial solution*, an object used in building up a collection of services for a given input expression. *Initial* partial solutions are created from individual services; otherwise partial solutions are created by combining other partial solutions. Each of these objects has the following components:

- **Expr**: clauses from the logical form that this solution does *not* cover. In the *empty solution* this field is equal to the input expression; in a *complete* solution this field is empty.
- **Cost**: the combined cost of all the component solutions making up this solution.
- **Input-links**: the mapping from variables in the logical expression to variables in the solution's service(s).
- **Output-links**: the mapping from variables in the solution's service(s) to variables in the logical expression.
- **Internal-partials**: the collection of initial (sometimes also called *primitive*) partial solutions from which this partial solution was constructed.
- **Matches**: the collection of *match* objects belonging to this partial solution. There is one match object for *each clause* matched by a service. See section 5.2.
- **Local-matches**: this field is currently unused, but is intended for use when optional system-dependent rewrites are introduced.
- **Goodness**: value based upon the sum of the cost of component solutions and other factors (e.g. communication cost between component solutions)

5.2 Matches

A *match* is an object built during pattern-matching. Matches are also used during later stages of processing, since they provide the link between a partial solution and its component services. Each match object comprises:

- the service, part of whose pattern was matched
- a "name", consisting of a list of the service and an instance number, used to distinguish different instances of the same service (for example, when one service matches two different parts of the same pattern -- see page 19)
- the clause in the input expression that was matched
- the clause in the service's pattern that did the matching
- the variable mappings produced by the match (see also *input-links* and *output-links*, above)

- embedded partial solutions created via recursive call(s) to the formulation algorithm; applicable only when the matched clause is a context, and therefore has internal subexpressions.
- embedded variable mappings associated with the embedded partial solutions.

For example, consider a service named *NAMEOF328*, whose pattern is ((NAMEOF <x> <y>)), and an input expression containing the clause *(NAMEOF ?JX1 "VINCENNES")*. The match object created by the pattern matcher will look like:

```
SERVICE:                    [Service: NAMEOF328]
VAR-MAPPINGS:               (("VINCENNES" . Y) (?JX1 . X))
PATTERN:                    ((NAMEOF <x> <y>))
CLAUSE:                     (NAMEOF ?JX1 "VINCENNES")
EMBEDDED-SOLUTIONS:         NIL
EMBEDDED-MAPPINGS:          NIL
```

As a more complex example, consider a service named *GENERAL-CARDINALITY*, belonging to the LISP server, that matches cardinality expressions. The query "How many C1 cruisers are there?" produces the request

```
(AND (IS-TERM
       #S(CONTEXT
            :OPERATOR CARDINALITY
            :FREE-VARS NIL
            :LOCAL-VARS NIL
            :OPERATOR-VAR ?JX1
            :CLASS-EXP ((IN.CLASS ?JX1 CRUISER)
                        (VESSEL-OVERALL-READINESS-OF ?JX1 C1))
            :STAT-VAR NIL
            :CONSTRAINTS NIL)
       ?JX5)
     (RESPONSE ?JX5))
```

The match object created by matching the *GENERAL-CARDINALITY* service to the first clause is

```
SERVICE:       [Service: GENERAL-CARDINALITY]
VAR-MAPPINGS:  ((<c.GENERAL-CARDINALITY.5> . ?JX5))
PATTERN:       ([Context-spec: CARDINALITY])
CLAUSE:
 (IS-TERM
   #S(CONTEXT
       :OPERATOR CARDINALITY
       :FREE-VARS NIL
       :LOCAL-VARS NIL
       :OPERATOR-VAR ?JX1
       :CLASS-EXP
        ((IN.CLASS ?JX1 CRUISER)
         (VESSEL-OVERALL-READINESS-OF ?JX1 C1))
       :STAT-VAR NIL
       :CONSTRAINTS NIL)
   ?JX5)
EMBEDDED-SOLUTIONS:  ([PS: 9.0] NIL)
EMBEDDED-MAPPINGS:
(((CRUISER . <w.CRUISER23.2>)
  (?JX1 . <x.CRUISER23.2>)
  (?JX6 . <y.NAMEOF328.3>)
  (?JX1 . <x.NAMEOF328.3>)
  (C1 . <y.VESSEL-OVERALL-READINESS-OF859.1>)
  (?JX1 . <x.VESSEL-OVERALL-READINESS-OF859.1>)
  ((?JX6) . <X.VALUE-EXISTS-RESPONSE.4>))
 NIL
 ((?JX1 . <X.GENERAL-CARDINALITY>)))
```

The expression embedded in the cardinality clause (*class-exp*) was solved recursively in the course of the matching.[8] The solution, which retrieves all the C1 cruisers, is the first element in the EMBEDDED-SOLUTIONS field, above. The second element in EMBEDDED-SOLUTIONS is NIL because there was no constraint expression -- had this context been, say, a universal quantification (*FORALL var class-exp constraints*), then this second element would have been the solution to the *constraints* expression.

5.3 The Formulation Algorithm

The formulation algorithm is shown below.

[8] Embedded solutions are computed only once, of course, regardless of how much pattern-matching goes on.

```
Find the set P of initial partial solutions for request expression.
Choose the "best" n elements p1,...,pn in P, and
  let focus set F = {p1,...,pn}
While no member of F is a complete solution
  Choose the n best elements f1,...,fn of F
  For each fi
    "Age" fi, reducing its goodness by some factor
    Choose the best element p from P that can be combined with fi
    Let fi' = combine-partials(fi,p)
    Add fi' to F
  If no member of F was expanded,
    Report that no solution was found
  Else
    Let F = the best n elements of F, plus complete solutions in F
```

Each *initial partial solution* is essentially an instance of a service whose pattern has been completely matched by part of the expression. These are found by first restricting the search to those services matching some predicate appearing in the pattern (recall the discussion of how services are indexed, in section 4), then doing more complete pattern-matching on the restricted set. The set of initial partial solutions includes all possible ways to match a service to a pattern. For example, given an expression

```
(AND (P ?JX1)          ;clause (1)
     (P ?JX2)          ;clause (2)
     (Q ?JX3))         ;clause (3)
```

and a service with the pattern

```
(AND (P <x>)
     (Q <y>))
```

there will be *two* initial partial solutions produced, one in which the service has matched clauses (1) and (2), and the other in which the service has matched clauses (1) and (3).

Beginning with a focus set of initial partial solutions, the formulation algorithm seeks to expand solutions in the focus set. "Aging" elements in the focus set -- that is, reducing their goodness by some small factor each iteration -- results in throwing out nonproductive solutions after a while. This algorithm is not complete -- it is possible that no member of the initial focus set will provide a successful starting point, one might consider adding a step in the *while* loop that allows new starting points to be added to the focus set. Nor does the greedy heuristic used guarantee finding the optimal solution.

Partial solutions can only be combined if there is a *connection* between them -- one variable provided as output by one solution must be desired as input by the other. The combination does not, however, cement these input/output links; establishing such data dependencies is the job of the execution planner (see section 6). After combining two partial solutions, the resulting partial solution's input is the *union* of the component solutions' inputs, and the outputs include any output provided by a component partial solution.

5.4 Formulation and Embedded Contexts

Notice that the formulation algorithm does not include recursive calls to itself. Calls to the formulation algorithm for handling embedded expressions (i.e., the class expression or constraint expression of a context) are made as part of the *pattern-matching* process (see section 5.2). It is necessary to find all the possible ways that the embedded expressions can be solved, *before* doing pattern-maching at the top level. When seeking services that will cover a context clause, the pattern matcher

1. Finds the set S of services whose pattern includes some context-specification whose operator matches this context's operator (e.g., CARD, FORALL)
2. Finds a collection[9] C_1 of solutions to the context's class expression by recursively calling the formulation algorithm on it
3. Finds a collection C_2 of solutions to the context's constraint expression, in the same manner
4. Takes the cross-product of C_1 and C_2, to form a set C' of *embedded* (or *internal*) solutions for the context clause. Each embedded solution represents one possible way to solve the embedded expressions.[10]
5. For each service in S and for each embedded solution in C', checks whether S matches the context clause *assuming* that particular embedded solution.

   Some services in S make use of the embedded solution information in matching elements in this set of pattern clauses (see the discussion of the pattern field in services, specifically the use of the :COVERS-OWNER parameter, in section 4). Other services do not use the embedded solution information for pattern-matching. In either case, of course, the embedded solution(s) will be used for execution planning.

6 Execution Planning

The job of the execution-planning phase is to take a complete solution produced by the formulation algorithm and produce an *execution plan* that makes use of the resources specified by the solution and provides a specification of the dataflow among those resources.

[9] Recall that the formulation algorithm produces n solutions.

[10] An exception to this is the *disjunction context*, in which the class expression field contains an arbitrarily long list of disjuncts, each of which must be solved recursively. If each disjunct can be solved in several ways, it is impractical to take the cross-product; instead, we create only one embedded solution, using the best solution for each disjunct.

6.1 Partial Execution Plans

An execution plan is a sequence of *partial execution plans*, each of which consists of the following

- **Owner**: the name of the server to which this partial plan belongs, and which will execute this partial plan
- **Inputs**: the inputs as identified by the Janus variables (prefixed by "?JX") in the request that this partial plan expects to receive.
- **Outputs**: the outputs (also identified by Janus variables) that this partial plan will produce
- **Wrapper**: a place to hold the data passed to this partial plan by other partial plans. The wrapper is an object that includes:
  - A list of *streams* -- pointers back to where the data came from
  - For each stream, a list of *labels* -- each label is a Janus variable from the request
  - For each stream, a *bucket* containing the actual tuples passed in as data

  For example, the wrapper for a partial execution plan (after a previous partial plan has been executed) might look like this:

```
#S(WRAPPER :STREAMS  (#S(PARTIAL-EXECUTION-PLAN...))
           :LABELS   ((?JX1 ?JX2))
           :BUCKETS  ((("VINCENNES" 1)
                       ("NIMITZ" 2)
                       ("FREDERICK" 1)...)))
```

  This wrapper has only received input from one place: the partial plan on the :STREAMS list. The data are in the form of pairs -- the first element in each pair is an instantiation of the Janus variable ?JX1 in the request, and the second element in each pair is an instantiation of the variable ?JX2.
- **Body**: the code to be executed by the server (i.e., by the server's *executor* function). This code is built by the server's *execution planner* function.

6.2 Partitioning and Establishing Dependencies

There are two processes in creating an execution plan from a complete solution: *partitioning* the solution according to server, and setting up the *dependencies* among the nodes in the partition, based upon the possible inputs and outputs of each node. There is a certain circularity here that makes the process difficult: one can not set up dependencies until the solution is partitioned according to server, yet in order to partition properly (for example, to split a node belonging to a particular server into two nodes in order to allow another node to fit between them) one needs to know what the dependencies are. The current implementation does not handle this issue particularly well: it partitions first, then attempts to set up dependencies.

Assuming that partitioning has been reasonably done, there is another problem of circularity, involving the expected inputs and outputs of each node: one would like to be able to operate from a global perspective, *using* the expected inputs and outputs to optimally plan dataflow links; on the other hand, at the level of each node, one would like the global planner to *provide* the desired inputs and outputs, so as to produce optimal code for this node of the execution plan. In essence, the global level says, "tell me what to expect," and the local level says "tell me what you

need!" The current implementation is limited as follows: planning at both levels assumes that all inputs are provided, and all outputs are required.

6.3 Servers' Execution Planners

As discussed in section 4, every server has associated with it an *execution planner* function. Execution planner functions take three arguments:

1. A list of Janus variables it can expect to have values for as input,
2. A list of Janus variables it should produce as output, and
3. The solution-object for the portion of the query that is to be handled. From the solution object, one can obtain the component services (and thus their method slots, which contain the necessary code fragments) and variable mappings from Janus variables to service variables.

These functions should return:

1. Code to be executed by the execution function (see section 7),
2. Reductions from tuples to single variables, if any (e.g., if the execution-planner determines that output variable ?JX3 should be treated as *(tuple ?JX1 ?JX2)* for information-passing, then the list *(?JX3 ?JX1 ?JX2)* is one such reduction), and
3. The Janus variables that the code will produce output for.

7 Execution

The execution phase takes an execution plan (i.e., a list of partial execution plans), and iterates through it sequentially:

```
For each partial execution plan p
  Combine the data from the streams in the wrapper of p
  Call the execution function for the owner of p
  Pass the output tuples (according to the dataflow links of p)
   into the wrapper objects of partial plans further on
Return the output provided by the last partial execution plan
```

7.1 Combining Data

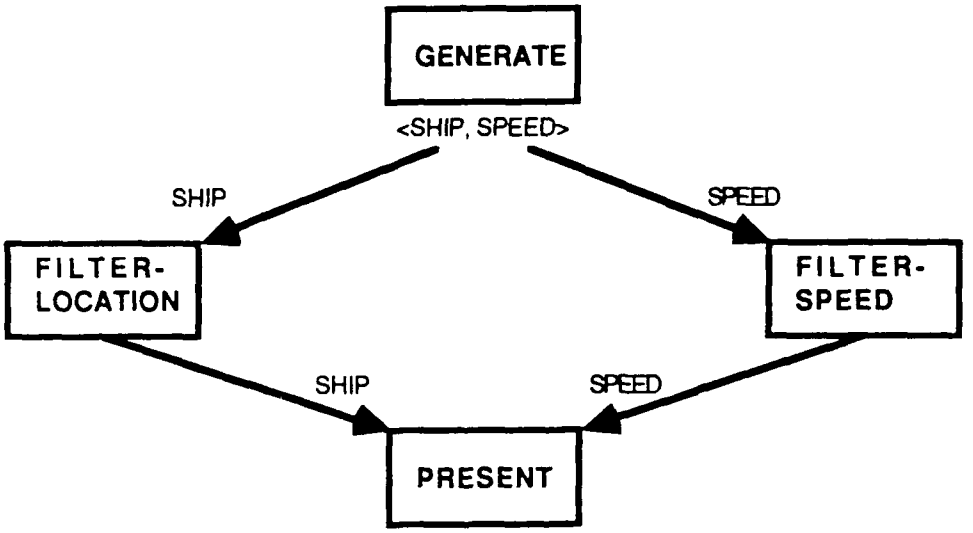
Previous approaches to the multiple systems problem (e.g., [4] and [3]) have assumed, for the purpose of execution, a straightforward dataflow model in which nodes accomplish execution and arcs are streams of values. Unfortunately, the problem of passing and combining data among multiple systems is more complex than this model will accommodate. In most cases, it is necessary to pass sets of *tuples* rather than sets of values, using a generalization of the *join* operation to combine data. There are problems that even this does not address.

7.1.1 Passing tuples, not values

Consider a scenario in which the user has requested a table of "the speeds of the ships in the Indian Ocean that are faster than 20 knots", and in which the resulting solution involves four services:

1. *Generate* generates pairs of ships and speeds.
2. *Filter-location* filters a list of ships according to whether or not they are in a given location.
3. *Filter-speed* filters a list of numerical speed values according to whether or not they are faster than a given speed, and
4. *Present* presents a table of ships and speeds.

In a model in which streams of values are passed, *generate* will pass a stream of ships to *filter-location*, which will pass a filtered stream of ships to *present*; *generate* will also pass a stream of speeds to *filter-speed*, which will pass a filtered stream of speeds to *present*.

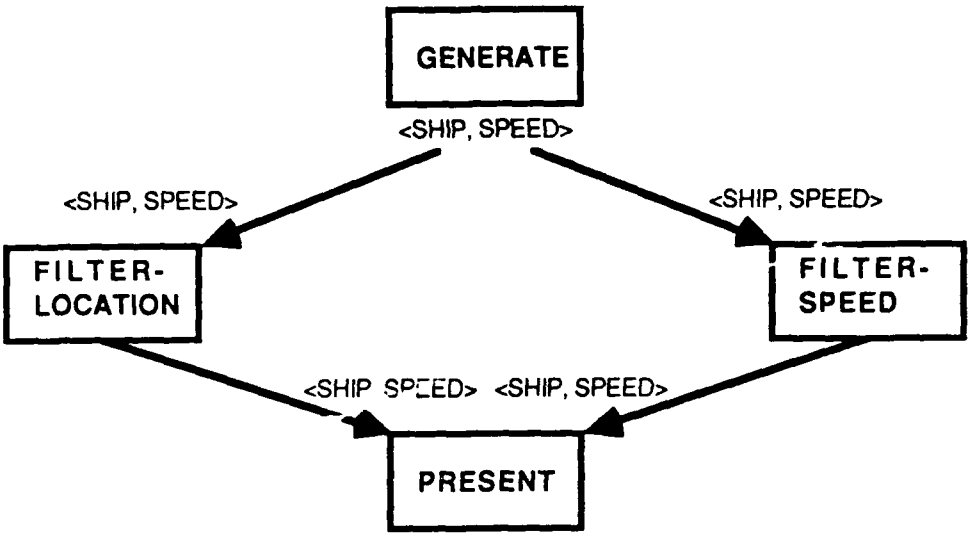


The problem is this: *present* has received both ships and speeds, but how can it now decide which speeds belong to which ships? The relation of ships to speeds was lost because, although pairs were generated by *generate*, they were split up in order to pass the data; once split up, there is no way to put them back together again.

An obvious approach to solving this problem is to pass not streams of *values*, but streams of *tuples* of values, never breaking up a tuple. This increases the volume of data passed, of course, but it does ensure that the appropriate relationships are maintained.

Now suppose that the same arcs represent the passing of tuples rather than of individual values. *Generate*

generates pairs <*ship, speed*> of ships and speed values. Because *filter-location* requires the ships, *generate* passes all the pairs to *filter-location*, which filters out those tuples in which the ship is not at the appropriate location. *Filter-location* then passes the filtered set of tuples to *present*. Because *filter-speed* requires the speed values, *generate* passes all the pairs to *filter-speed*, which filters out those tuples in which the speed is too slow. *Filter-speed* then passes the filtered set of tuples to *present*.



The situation has improved, in that we have maintained the association between ships and speeds. However, *present* has now received two *different* sets of ship-speed pairs, one filtered according to a property of the ship, and the other according to a property of the speed. How do we combine them?

7.1.2 Join and Cross-join

A solution that works in many cases is a database *join* across the attributes that the streams have in common.[11] A *join* effectively takes the cross-product of the incoming sets of tuples, and then removes from the cross-product any tuples in which the values of the common attributes are not equal. For example, the *join* of a set of tuples <*ship, location*> with a set of tuples <*ship, speed*> across the attribute *ship* will result in a set of tuples <*ship, location, speed*>, which will include only values of *ship* that appeared in *both* sets of tuples being joined.

It is quite possible that two (or more) streams of input will have *no* attributes in common, and in such cases *join* can not be used. In such a situation, one would like to use a version of the join operator that computes the cross product, but -- because zero attributes are held in common -- does not attempt to do the filtering operation.

[11] In this case, both attributes -- ship and speed-value -- are in common, so the join is just the intersection of the two sets of tuples.

For this reason, we use a combining operator called *cross-join*: when the incoming streams have attributes in common, *cross-join* is equivalent to *join*; when there are no common attributes, *cross-join* is equivalent to the cross-product.

7.1.3 Problems with *cross-join*

In some cases, the strategy adopted by *cross-join* is not appropriate. For example, suppose the user has requested the commanders and destinations of all the ships, and that one server generates pairs of <*ship, commander*> while another generates <*ship, destination*>. Suppose, further, that each of these servers has only *incomplete* information so that each produces tuples about some ships not known by the other.

SHIP (?jx1)	COMMANDER (?jx2)	SHIP (?jx1)	DEST (?jx3)
VINCENT	SMITH	VINCENT	HAWAII
FOX	JONES	NIMITZ	HAWAII
FREDERICK	BROWN	FREDERICK	SAN DIEGO

In such a case, the *cross-join* operation will recognize that the *ship* attribute is held in common by the incoming sets of tuples, and thus combine the sets of tuples using *join*:

SHIP (?jx1)	COMMANDER (?jx2)	DEST (?jx3)
VINCENT	SMITH	HAWAII
FREDERICK	BROWN	SAN DIEGO

Notice that as a result of *join*'s filtering operation, *FOX* and *NIMITZ* do not appear in the output data. Considering the fact that the user requested the commanders and destinations of *all* the ships, this is an undesirable result: it is likely that the user wants to see whatever information is available about each ship even if that information is incomplete.

This example serves to illustrate that *cross-join* is not appropriate in all instances. Unfortunately, there is currently no easy way to identify such cases. For the present time, *cross-join* is always used to combine data.

7.2 Servers' Execution Functions

The execution function (or *executor*) for a server is a function taking the following arguments:

1. A list of tuples representing input values,
2. A sequence of Janus variables identifying the tuple elements, and
3. Code produced by the execution planner.

The execution function should return two values:

1. A list of tuples representing output values, and
2. A sequence of Janus variables identifying the tuple-elements.

8 Status and Extensions

8.1 Experience

The MUS component described in this documentation has been successfully implemented and used in the domain of the Fleet Command Center Battle Management Program (FCCBMP), using an internal version of the Integrated Database (IDB) -- a relational database -- as one underlying resource, and a set of LISP functions as another. The system includes more than 800 services, and produces an execution plan for a typical request in seconds or fractions of seconds; it also reports failure to create an execution plan within seconds (for example, in cases where no service exists covering part of a request expression). Queries handled include those involving negation of simple predicates, existential and universal quantification, cardinality, and the most common disjunctions,[12] as well as queries that are simply conjunctions of clauses. Both queries requesting values (actually, tuples of values are returned) and yes/no queries are handled.

An earlier version of the system described here was successfully used within an expert system project, in which Janus provided natural language access to data in Intellicorp's KEE knowledge-base system to objects representing hypothetical world-states in a simulation system and to LISP functions capable of manipulating this data.

8.2 Limitations and Extensions

There are several limitations and possible extensions in the current implementation of the system:

1. Although underlying-system-*independent* rewrite rules are supported, underlying-system-*dependent* rewrites are not. In order to allow these, it is necessary to modify the formulation algorithm so that, rather than expanding an intermediate partial solution, one can modify it by applying a system-dependent rewrite. One would need to avoid combining partial solutions evolved from different request expressions, as could be the case if different rewrites have applied.
2. The formulation algorithm is not complete: it is possible that none of the initial partial solutions chosen as starting points can be expanded into a solution that covers the entire request. One might consider using a different search algorithm (e.g., a modification of the A* search employed by [3]) or modifying the algorithm to make it complete.
3. The model of service costs, assuming a single scalar cost value, is too simple for many likely real-world situations. A better model would distinguish aspects of cost like the reliability of the service's data, the cost of communicating with it, and the service's time and space requirements.
4. The current execution model assumes that the representations of entities are the same in different underlying systems. This is a severe limitation and should be addressed as soon as possible. There are three immediately apparent cases:
    - Spelling variations. For example, one system may store the name of a ship as the "CARL VINSON" while another stores it as "VINSON C".

[12] Those expressing membership in a set, e.g., (OR (EQ x item1) (EQ x item2) ...).

- Different internal representations. For example, one system may store "the ship itself" as an ID number (e.g. "0123") while another uses the ship's name (e.g. "VINSON C").
- Differing data decompositions. For example, one system may store dates as single values (e.g. date="07-13-65") while another stores them as several values (e.g. month=07, day=13, year=65).

5. Error recovery. A single plan is created in response to a request, and if it fails, the system has no recourse but to report an error. It should be possible to modify the formulation and execution-planning phases of processing to allow the creation of alternative plans. Notice that this still does not address the *reasons* for plan failure: if a disk error has been encountered while making use of a resource, it makes no sense to try again with a different plan that requires the same resource.

References

[1] Garey, M. R. and D. S. Johnson. *Computers and Intractability: A guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[2] Hinrichs, E., D. Ayuso, and R. Scha. The Syntax and Semantics of the JANUS Semantic Interpretation Language. In R. Weischedel, D.Ayuso, A. Haas, E. Hinrichs, R. Scha, V. Shaked, D. Stallard (editors), *Research and Development in Natural Language Understanding as Part of the Strategic Computing Program*, chapter 3, pages 27-34. BBN Laboratories, Cambridge, Mass., 1987. Report No. 6522.

[3] Kaemmerer, W. and J. Larson. A graph-oriented knowledge representation and unification technique for automatically selecting and invoking software functions. In *Proceedings AAAI-86 Fifth National Conference on Artificial Intelligence*, pages 825-830. AAAI, Morgan Kaufmann Publishers, Inc., 1986.

[4] Pavlin, J. and R. Bates. *SIMS: Single Interface to Multiple Systems*. Technical Report ISI/RR-88-200, ISI, February, 1988.

[5] Stallard, David. Answering Questions Posed in an Intensional Logic: A Multilevel Semantics Approach. In R. Weischedel, D.Ayuso, A. Haas, E. Hinrichs, R. Scha, V. Shaked, D. Stallard (editors), *Research and Development in Natural Language Understanding as Part of the Strategic Computing Program*, chapter 4, pages 35-47. BBN Laboratories, Cambridge, Mass., 1987. Report No. 6522.

[6] Stallard, David. A Manual for the Logical Language of the BBN Spoken Language System. July, 1988.